Log-Only Temporal Object Storage[*]

Kjetil Nørvåg and Kjell Bratbergsengen
Department of Computer and Information Science
Norwegian University of Science and Technology
7034 Trondheim, Norway
{noervaag,kjellb}@idi.ntnu.no

Abstract

As main memory capacity increases, more of the database read requests will be satisfied from the buffer system. Consequently, the amount of disk write operations relative to disk read operations will increase. This calls for a focus on write optimized *storage managers. In this paper we show how the Vagabond object storage manager uses no-overwrite sequential writing of long blocks to achieve high write performance. Vagabond also supports versioned/temporal objects, with the no-overwrite policy used, this does not imply any extra cost. Large objects, e.g., video and matrixes, are divided into large chunks. This makes it easy to achieve high read and write bandwidth. This is important, since in many application areas, high data bandwidth is just as important as high transaction throughput. The buffer system in Vagabond is object based, rather than page based. This gives better utilization of main memory. Transparent compression of objects on disk is supported.*

1 Introduction

The main bottleneck in a well designed database system is usually secondary storage access. In a database system, most accesses to data are read operations. Consequently, database systems have been *read optimized.* However, as main memory capacity increases, the amount of disk write operations relative to disk read operations will increase. This calls for a focus on *write optimized* database systems. Another aspect that gives this issue increased importance, is database systems which also needs high performance in terms of data bandwidth, and not only in transaction throughput (although these points are related). This is especially important for new emerging application areas, like video database systems and super computing applications, which have earlier used file systems. These applications will have a need for high data bandwidth.

Increasing the effective bandwidth can be done in three ways: 1) by reducing seek time, 2) by reducing the amount of time spent in rotational delay, and 3) parallel I/O. This can be achieved by sequential writes, large blocks [2], and the use of RAID. In current database systems, the first two are partially achieved by the use of write ahead logging (WAL), which defers the non-sequential writing. However, sooner or later, the data has to be written to the database. This involves the writing of lots of small objects, almost always one access for each individual object. Our solution to this problem, is *to eliminate the current database completely*, and use a *log-only* approach. The log is written contiguously to the disk, in a no-overwrite way, in large blocks. It is obvious that this gives optimal write performance, but possibly at the expense of read performance. The question is: is it possible to design such a system, which is able to operate within database constraints, using a log written this way? We will in the following show that this is indeed possible, in our description of our temporal object storage manager Vagabond[1].

The rest of the paper is organized as follows. In Section 2, we discuss related work. Section 3 gives an overview of the Vagabond storage manager, and in Section 4 we present some of the advantages that motivated the design of Vagabond. In Section 5 we describe the object index structures used. In Section 6 we describe how objects are stored in Vagabond, and

[1]From *Webster's Encyclopedic Unabridged Dictionary*: Vagabond: "a person, usually without a permanent home, who wanders from place to place; nomad". Quite similar to our objects!

in Section 7 we describe the physical storage structures. In Section 8 to 11 we describe briefly transaction and recovery related issues. Finally, in Section 12, we present future work and conclude the paper.

2 Related Work

A no-overwrite strategy has been used in shadow-paging recovery schemes earlier, e.g., in System R [1]. Log-only database techniques are used in Postgres [6], an object-relational database system. Because of the high cost of commit in Postgres, the techniques did not gain any success at that time. The ideas from Postgres were borrowed and used in log structured file systems (LFS), first presented by Rosenblum and Ousterhout [4], later refined by Seltzer [5]. In a LFS, file and directory information are interleaved in a log. Our object storage management is based on LFS techniques, but with some major differences: support for a wide range of data granularities (very small as well as very large objects), transactional ACID properties, efficient support for distributed transactions, various access methods, and indexes. Other major differences are the number of objects in an object-oriented database system (OODB), which will usually be much larger than the number of files in a file system. While it is possible to have most of the file directory cached, this will not be possible for the object directory. Also, access patterns are radically different.

3 Overview of Vagabond

Most current database systems have a current database, and a log. The buffer system is page based, and data is read and written pagewise to the current database. The systems employ WAL, redo/undo-records are written in the log before data is modified in the database. By doing this, only log records have to be flushed at commit time. This saves a lot of disk bandwidth, since the WAL-data is usually much smaller than a page. Writing to the log also helps in getting a sequential write pattern. The drawback is that during recovery, possibly large parts of the log has to be read to make the database consistent. Also, when the buffer is full, and pages have to be replaced, random writes are necessary to write these back at their positions in the database.

Our storage manager takes this to the extreme: we do not have the current database, *we have only got the log*. Interleaved in the log is also the index structures necessary to retrieve data efficiently (but they are not necessarily clustered). Already written data is never modified, new versions of the objects are just appended to the log. Writes are always done sequentially, with large write block sizes. This is done by writing many objects and index entries, possibly from many transactions, in one write operation. Logically, the log is an infinite length resource, but the physical disk size is, of course, not infinite. We solve this problem by dividing the disk into large, equal sized, physical segments. When one segment is full, we continue writing in the next available segment. As data is deleted or migrated to tertiary storage, old segments can be reused. Deleted data will leave behind a lot of partially filled segments, the data in these near empty segments can be collected and moved to a new segment. This process, which is called *cleaning*, makes the old segments available for reuse. By combining cleaning with reclustering, we can get well clustered segments.

Instead of using a page buffer as is common in other systems, we use an object buffer, which is more efficient when data is not guaranteed to be well clustered. As the buffer fills up, it might be necessary to write dirty objects to disk. This is done according to some replacement strategy, usually a variant of LRU. Objects can be written before the transaction commits (steal strategy), this avoids force of large amounts of data at commit time. Index blocks, however, are never updated before the transaction commits. Commit of a transaction is done by writing the index entries atomically to disk. The index blocks themselves, which consists of many possibly unrelated entries in each, can be written later, since they can be reconstructed at recovery time from the written index entries. This simplifies abort, and is especially important in a client-server environment. It is also useful to exploit optimistic concurrency control.

Recovery in a log-only database like Vagabond can be done very fast. At regular times, a checkpoint operation is performed, and at recovery time we just do an analysis pass from the last known checkpoint to the end of the log (where the crash occurred).

In a traditional system with in-place updating, keeping old versions of objects usually means that the previous version have to be copied to a new place before update. This doubles the write cost. In Vagabond, this is not necessary. Keeping old versions come for free (except for the extra disk space), only an additional index is needed. Thus, our system supports temporal database systems in an efficient way.

4 Advantages and New Opportunities

Because the log-only, no-overwrite approach, is radically different from the techniques used in current systems, it is appropriate to describe the advantages of such an approach. In the rest of this section, we will take a closer look on some issues which are affected.

Objects vs. Pages. Most current storage managers are page servers. We can liberate ourselves from the page server based earlier approaches. There is no inherent reason why pages would be better suited. It is also easier to guarantee larger segment sizes on large objects. This is important in, e.g., video server applications.

Transparent Compression of Data. Since objects are not written to the same physical disk block(s) every time, the compression ratio and storage size might change.

Easier On-Line Backup. The written segments are time stamped, and with a no overwrite strategy, it is enough to know the last time of backup to know where backup should be started now. Backup could also be done on-line, and again, even if we stop backup when the load is high, we know where to continue in less busy periods.

Flash Memory. Very high performance can be achieved if we use fast non-volatile memory instead of disk. One example of such an storage technology is flash memory. Flash memory is byte readable, and fast, but write/erase has to be done blockwise. This facilitates a storage strategy with no in-data modifications.

Write-Once Memory. With write-once storage, there is a need for a no-overwrite strategy.

RAID Technology. Disk access times and bandwidth improves at a much lower rate than main memory, and parallel disk systems are necessary to get high performance. To benefit from RAID technology, the write blocks has to be much larger than those used in traditional systems. In addition, in normal systems, sequential writes are only about 3-5 times faster than random writes, while in RAID, sequential writes can be up to 20 times faster [7].

Super Computing Applications. In many super computing applications, computations are done on large matrixes and arrays. To be able to do operations on these large structures, it is often necessary to break them into chunks which can be processed independently. It is necessary to retrieve and store these chunks efficiently. Until now, only file systems have been able to offer the performance necessary. However, there is a demand for some of the services offered by database systems in these areas: access control, concurrency control, and recovery. Performance close to file system performance is necessary to be applicable.

Non-Set-Based Access Patterns. Current relational database systems are optimized for operations on large sets of objects (tuples). As soon as the access pattern differs from this, or is navigational, it is difficult to get good performance, since applications can not benefit from automatic storage clustering. With our approach, we have more flexibility in reorganizing data. Reorganization can, e.g., be based on previous access patterns to the objects. This is especially interesting for applications with complex data structures, as is common in typical OODB applications.

Group Commit. Group commit, in addition to giving us larger writes, also gives opportunity for more intelligent clustering of objects from different transactions.

Fast Crash Recovery. The log-only approach is really a refined form of shadow storage. While this has its deficiencies, it also has a very nice and interesting feature: very fast crash recovery. By never updating in-place, recovery issues can be solved much easier.

Temporal Database Systems. Realizing a temporal database system is easy with our approach. Versioning comes at virtually no extra cost, while in a conventional database system versioning doubles the amount of data that has to be written.

Cache Coherence. Versioning/timestamping can be exploited in cache coherence in client-server environments, as is done in BOSS [3].

With all the advantages listed above, one might wonder why the idea of a log-only database system has not been brought into reality. All the nice features listed should have made them the preferred system type. The most important reason for this, is probably some problems encountered in the System R and Postgres projects.

The shadow paging scheme in System R, and the log in Postgres, both held the risk of declustering relations. Clearly, this can also be the case with our approach. However, we expect that the increased amount of main memory will compensate for the lack of clustering. Important is also that the access pattern is supposed to be quite different in our applications: more direct, navigational, and temporal accesses.

5 Object Identifier Index Structure

An object in an OODB is uniquely identified by an object identifier (OID). OIDs can be *physical* or *logical* (surrogate). If physical OIDs are used, the disk blocks where the objects resides is given directly from the OID, if logical OIDs are used, it is necessary to use a map to convert from logical OID to physical location. In our system, logical OIDs are necessary, since the objects are never written back to the same place. This might seem like an disadvantage, but even though a physical OID has a potential performance benefit, it also has a major drawback: relocation and migration of objects is difficult. Therefore, a logical OID is generally the preferred choice.

The data structure containing the necessary information to map from a logical OID to a physical location is in Vagabond called a *object descriptor* (OD). Each version of an object has its own OD, and the OD also contains other object specific information like create time, object size, and a class tag. *Create time* is the commit time of the transaction creating this version. The *class tag* identifies the class the object belongs to. This is necessary to efficiently support hierarchical concurrency control techniques. In Vagabond, collections and indexes are also stored as objects. In this case, the class tag is used as an index class identifier as well. This make the system very extensible, new index classes can be added to the system as needed.

The number of ODs can be very large, and a fast and efficient index structure is needed, an *OID index* (OIDX). In the OIDX, the ODs are stored in the leaf nodes. The OIDX is stored in the segments, interleaved with the data.

Efficient indexing in a log-only system is not trivial. Consider a tree-like index structure: if we update an index node, this will be written to a new location. The pointer in its parent node becomes invalid. The parent node needs to be updated, and this cascades up to the root. We also have another problem. Index blocks as well as objects may be relocated during cleaning or migration. In this case, structures having pointers to this data needs to be updated. To avoid inefficient structures, and reduce cascading updates, design of the index structure needs careful attention.

Fortunately, an OID index has some special properties that can be employed to make it more efficient. The keys in the index, the OIDs, are not uniformly distributed over a domain as keys commonly are supposed to be. If we assume the unique part of an OID to be an integer, new OIDs will always be assigned monotonic increasing values. If an object is deleted, the OID will never be reused. The access pattern is also important, it will always be a perfect match search, range search of OIDs is not interesting.

Each version of an object has its own OD, and this versioning complicates the index considerably, access to current as well as old versions of objects has to be supported by the index. However, for most applications, *most queries will be against the current data.* It is important that this access is as efficient as possible. The index structure in Vagabond is an ISAM variant, with extensions for the OD versioning.

6 Object Storage in Vagabond

In Vagabond, all objects smaller than a certain threshold are written as one contiguous object (not segmented into pages as is done in other systems). This threshold is configurable, but it should at least be in the order of 64 KB. Objects larger than this threshold, are segmented into subobjects. There are several reasons for doing it this way: 1) to avoid large objects blocking all other transactions when they are written to disk, 2) a segmented object is useful later, when only a small part of the object is to be read or modified, and 3) subobjects can reside on different physical devices, possibly on different levels in the storage hierarchy.

Often, only a small part of an object is changed when a new version is written. In this case, much can be gained if only *the changes* are written. This is especially the case if an object is a hot spot. A version which only contains the changes from last version, is called a *delta object.* The delta object itself can be made at low cost by the use of XOR between the new and the old version, and run-length encoding the result. At buffer replacement time, we should write the complete objects to disk. If we do not do this, object retrieval will be inefficient. We would have to retrieve the last written *complete* version, in addition to all delta objects written since the last complete write, to reconstruct the object.

To further reduce storage space, and disk bandwidth, objects can be compressed before they are written. In many application areas, e.g., statistical and scientific databases, a large number of NULL-fields exists in the records/objects. Without even knowing the structure of the objects, it is easy to run-length encode these objects. Compression/decompression is transparent to the applications, and a retrieved object will be decompressed before it is delivered.

Large objects in Vagabond are actually temporal index structures, where the segments of the objects are indexed on position range and time. The size of a subobject is always an integral number of disk sectors.

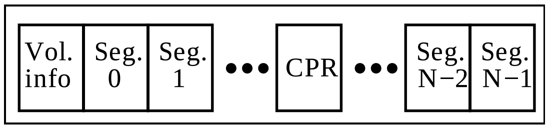

Figure 1: Volume structure.

7 Storage Structures

The log is stored on a *volume*. One volume is a configuration of one or more storage devices. The storage devices are typically disk partitions or volume files. A volume consists of a volume information block, a number of equal sized segments, and two or more checkpoint regions (CPR), as shown on Figure 1. The segment size has to be a tradeoff between different, partly conflicting, goals: to improve write efficiency, it is desirable that the segments written are as large as possible. On the other hand, large segments can make response time longer, since we have to wait for transactions during group commit, or, alternatively, result in a larger number of subsegments,writing a segment is done by writing one or more subsegment. Based on experiences from LFS, we expect that the segment size should be in the order of 1 MB for a single disk volume, but larger for parallel disk systems.

The volume information block holds static volume information, and is only read when the system is started. It is written when the volume is formatted, and when new devices are chained to the volume. Devices can be added to the volume while the system is running. Each device has its own volume information block. The volume information blocks are identical, except for the pointers to previous and next device, and contains information like segment size, number of segments, large object threshold, and location of segment status blocks.

The segments contains the objects and the index. We do not always have a full segment of data to write, and therefore we write data as subsegments. Subsegments are a number of fixed size disk sectors, and consist of a subsegment header, followed by the objects, ODs, index blocks, and transaction control information (e.g., prepare, commit and abort). For every object written to a segment, the corresponding OD is also always written in the same segment. The index blocks themselves can be written lazily to disk. In this way, we avoid having to flush all index blocks at commit time. If the system crashes, dirty index blocks that had not been written before the crash can be reconstructed from the ODs in the log. Having the ODs together with the objects also makes cleaning more efficient, we do not have to search an index to find out which objects are stored in a particular segment.

Information about the segments' status is kept in main memory during normal operation, in the *segment status table* (SST). A segment can be in the states *clean*, *dirty*, *alive*, or *current*. The segment we currently write to, is in the current state, segments containing non-deleted data (objects and OIDX blocks) are dirty. Segments written after the last checkpoint are alive. Segments with no data are clean. This means that data has never been written to it, all data previously residing on it has been deleted, or the segment has been cleaned (data moved to another segment, implicitly deleted from the old segment). We keep some statistics for each segment, the number of live bytes, number of read accesses, and last access time, to help us decide which segments to clean.

A checkpoint is finished by writing the SST and the location of the last written OIDX root block, to a checkpoint region. There are (at least) two checkpoint regions in a volume. These regions contain timestamps, so that during recovery, the last successfully written regions will be used. If the system crashes during the update of one of the regions, the two timestamps in the region will differ, and we use the previous region.

8 Segment Cleaning.

A segment starts in a *clean state*, it contains no data. A segment has to be clean before we can write to it, to maintain the no-overwrite policy. During writing of subsegments to a segment, the segment is *current*. When the segment is full, we start writing to a new segment. The new segment now goes from a *clean* state, to *current*. The previous segment is now *alive*, it contains valid data. When checkpoint is done, all previous segments in the alive state changes state to dirty.

As times go by, non-versioned objects gets deleted, and versioned objects are modified. The result is that more and more of the space in the segments is occupied by data which can be removed in its entirety (deleted non-versioned objects), or moved to tertiary storage (old versions). The disk will eventually fill up, and we do not have any clean segments left to write to. Before this happens, we have to move non-deleted objects in (almost empty) old segments, to the current (alive) segment, and if desired, old versions to tertiary storage. This process, which results in a segment going back to a clean state, is called cleaning. This process is also an excellent opportunity to *recluster* the database. Related data can be clustered together.

9 Transaction Management

When a transaction commits, it is necessary to write enough information in the log to be able to do recovery in case of crash. We have two options here: write the dirty object in its entirety, or only write a delta object. In Vagabond, we write delta objects if some given criteria is satisfied, e.g., based on the difference between the size of the object and the delta object, or storage format. In this way, we should be guaranteed that delta objects will only be written if it is beneficial. During normal operation, transactions will usually be able to commit much faster. This is especially important for objects that are hot spots.

Objects can be written to disk before a transaction commits, e.g., due to insufficient buffer space. However, it does not become visible before commit, which is done by updating the indexes. To avoid having to force the OIDX blocks to disk, only the index entries (in this case, the object descriptors), are forced to disk.

10 Checkpointing.

Efficient checkpointing is important. Currently, we have only sharp checkpointing in our design, but other strategies appropriate for our system will be considered. In general, sharp checkpointing is not a good alternative, since the system has to stop the normal processing while doing the checkpoint, but in Vagabond, the only restriction is that ongoing commit operations have to finish before the checkpointing starts, and new commit operations have to be delayed until the checkpointing finishes. Unlike conventional systems, the checkpoint operation does not involve lots of random writes. Data and index blocks are written to the log as usual, and the checkpointing is finished by writing the status information to one of the checkpoint regions.

11 Recovery

Crash recovery can be done fast and efficient in Vagabond. If crash recovery is needed, we start reading from the last checkpoint, and build the relevant structures in memory in one forward analysis pass. If writing has to be done because of insufficient buffer capacity, this is done in a new, clean, segment. Since we do not do any in-place update, it is easy to guarantee recovery idempotence. When we read a subsegment that was only partially written, we have come to the place where the system crashed. We do a checkpoint, and only after the checkpoint process is finished, the checkpoint regions are updated. Media failure in a log-only system can be handled using RAID or disk shadowing.

12 Conclusions and Future Work

In this paper, we have described the principles behind Vagabond, a write optimized storage manager. This system is log-based, with indexes embedded into the log. This will give very good write performance. We also expect that the read performance will be acceptable, even in the case where most of the data is *not* in main memory. This is especially true for applications with navigational access patterns, or many accesses to large objects.

The design of Vagabond is now finished, and we are currently in the implementation phase. In addition to implementing the storage manager as described in this paper, we will pursue issues related to super computing database applications (especially storage of matrixes), geographical information systems (which can exploit our support for temporal/versioned objects), and efficient buffer structures in the context of the object buffer in Vagabond. Vagabond is intended to be part of a parallel OODB, and we plan to exploit the features in this system.

References

[1] M. M. Astrahan et.al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems*, 1(2), 1976.

[2] K. Bratbergsengen and K. Nørvåg. Improved and Optimized Partitioning Techniques in Database Query Processing. In *Proceedings of the Fifteenth British National Conference on Databases, BNCOD15*, 1997 (to appear).

[3] D. E. Langworthy and S. B. Zdonik. Extensibility and Asynchrony in the Brown-Object Storage System. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice Hall, 1996.

[4] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, 1991.

[5] M. Selzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter 1993 Conference*, 1993.

[6] M. Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of the 13th Conference on Very Large Databases*, 1987.

[7] M. Stonebraker. *Readings in Database Systems (2nd edition)*. Morgan Kaufmann, 1994.